StreamNative

vs. vs.

Apache Pulsar™ vs. RabbitMQ™ vs. NATS JetStream: A Comprehensive Benchmark Report of Messaging Platforms

Jihyun Tornow, Director of Product Marketing, StreamNative
Matteo Merli, Apache Pulsar PMC Chair, CTO, StreamNative
Elliot West, Sr. Platform Engineer, StreamNative
Alice Bi, Content Strategist, StreamNative
Kristin Crosier, Content Marketing Manager, StreamNative

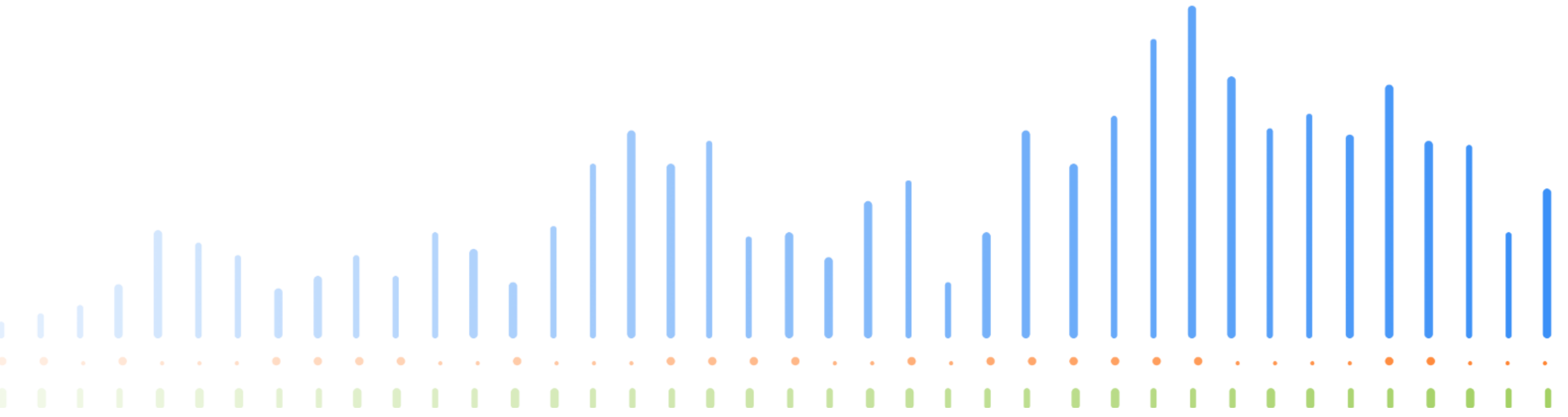

Table of Contents

Executive Summary

When building scalable, reliable, and efficient applications, choosing the right messaging and streaming platform is critical. In this benchmark report, we compare the technical performances of three of the most popular messaging platforms: Apache Pulsar^{TM}, RabbitMQ^{TM}, and NATS JetStream.

The tests assessed each messaging platform's throughput and latency under varying workloads, node failures, and backlogs. Please note that Apache Kafka was not included in our benchmark as Kafka does not support queuing scenarios. For more information on Kafka, please refer to the Pulsar vs. Kafka 2022 Benchmark report.

Our objective was to provide guidance on each platform's capabilities and reliability, and help potential users choose the right technology for their specific needs. The results of these tests provide valuable insights into the performance characteristics of each platform and will be helpful for those considering using these technologies.

Key Benchmark Findings

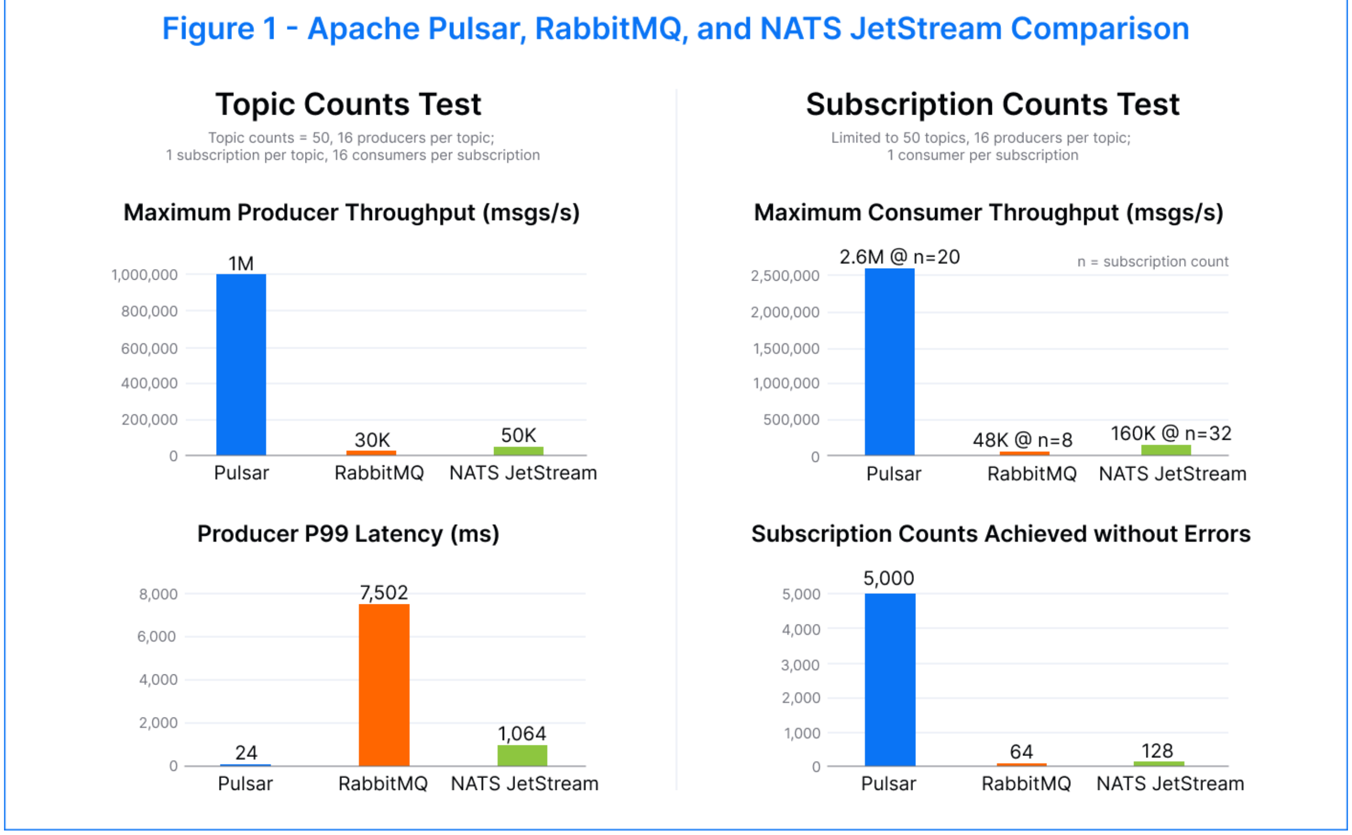

Figure 1 - Apache Pulsar, RabbitMQ, and NATS JetStream Comparison

- Throughput:
    - Pulsar showed a higher peak consumer throughput of 2.6M msgs/s compared to RabbitMQ's 48K msgs/s and NATS JetStream's 160K msgs/s.
    - Pulsar was able to support a producer rate of 1M msgs/s — 33x faster than RabbitMQ and 20x faster than NATS JetStream.
- Backlog:
    - Pulsar outperformed RabbitMQ during the backlog drain with a stable publish rate of 100K msgs/s, while RabbitMQ's publish rate dropped by more than 50%.
- Latency:
    - Pulsar's p99 latency was 300x better than RabbitMQ and 40x better than NATS JetStream at a topic count of 50.
- Scalability:
    - Pulsar achieved 1M msgs/s up to 50 topics and provided a publish rate above 200K msgs/s for up to 20K topics.
    - RabbitMQ was able to process 20K msgs/s, and NATS was able to support 30K msgs/s for topic counts up to 500.

Background

Before we dive into the benchmark tests, let's start with a brief overview of the architecture, features, and ideal applications for each messaging platform.

Apache Pulsar

Apache Pulsar is an open-source, cloud-native messaging and streaming platform designed for building scalable, reliable applications in elastic cloud environments. Its multi-layer architecture includes multi-tenancy with resource separation and access control, geo-replication across regions, tiered storage, and support for five official client languages. These capabilities make Pulsar an ideal choice for building applications that require scalability and reliability.

One of the standout features of Pulsar is its shared subscription, which is handy for queuing applications and natively supports delayed and scheduled messages. Additionally, Pulsar simplifies application architecture by supporting up to 1M unique topics, making it widely used for high-performance data pipelines, event-driven microservices, real-time analytics, and other real-time workloads. Originally developed at Yahoo! and committed to open source in 2016, Pulsar has become popular among developers and leading organizations.

RabbitMQ

RabbitMQ is a popular and mature open-source distributed messaging platform that implements the Advanced Message Queuing Protocol (AMQP) — often used for asynchronous communication between services using the pub/sub model. The core of RabbitMQ's architecture is the message exchange, which includes direct, topic, headers, and fanout exchanges. RabbitMQ is designed to be flexible, scalable, and reliable, making it an effective tool for building distributed systems that require asynchronous message exchange.

RabbitMQ is a good choice if you have simple applications where message durability, ordering, replay, and retention are not critical factors. However, RabbitMQ has limitations in dealing with massive data distribution and may not be suitable for applications with heavy messaging traffic. In addition, the platform does not support other messaging patterns such as request/response or event-driven applications.

NATS JetStream

NATS is an open-source messaging platform optimized for cloud-native and microservices applications. Its lightweight, high-performance design supports pub/sub and queue-based messaging and stream data processing. NATS JetStream is a second-generation streaming platform that integrates directly into NATS. NATS JetStream replaces the older NATS streaming platform and addresses its limitations, such as the lack of message replay, retention policies, persistent storage, stream replication, stream mirroring, and exactly-once semantics.

NATS utilizes a single-server architecture, which makes it easy to deploy and manage, particularly in resource-constrained environments. However, NATS does not support message durability and may not be suitable for applications that require this or complex message routing and transformations. Despite this, NATS offers an asynchronous, event-driven model that is well-suited for simple pub/sub and queue-based messaging patterns due to its high performance and low latencies.

Overview of Tests

What We Tested

We conducted four benchmark tests to evaluate each platform's performance under various conditions, such as workload variations, node failure, and backlogs. The aim was to assess each platform's responses to these conditions and to provide insights into their capabilities in a given environment.

1. Node failure

Failures will inevitably occur in any platform, so it's vital to understand how each platform will respond to and recover when such events occur. This test aimed to evaluate the performance of each platform in response to a single node failure and subsequent recovery. To simulate a node failure, we performed broker terminations and resumptions via `systemctl stop` on the node. We then monitored the performance of the remaining nodes as they took on the workload of the failed node. We anticipated a decrease in producer throughput and an increase in producer latency upon failure due to the overall reduction in the cluster's resources.

2. Topic counts

This test examined the relationship between peak throughput and latency and the number of topics within a platform. We measured the performance of each platform at various topic counts, from very small to very large, to understand how the platform's performance changed as the number of topics grew. We expected that for very small topic counts, the platform would exhibit sub-par performance due to its inability to utilize available concurrency effectively. On the other hand, for very large topic counts, we expected performance to degrade as resource contention became more pronounced. This test aimed to determine the maximum number of topics each messaging platform could support while maintaining acceptable performance levels.

3. Subscription counts

Scaling a messaging platform can be a challenging task. As the number of subscribers per topic increases, changes in peak throughput and latency are expected due to the read-amplification effect. The imbalance between writes and reads occurs because each message is read multiple times. Despite this, we would expect the tail reads to be relatively lightweight compared to the producer's writes, which are most likely coming from a cache. Increased competition among consumers to access each topic may also lead to a drop in performance. This test aimed to determine the maximum number of subscriptions per topic that could be achieved on each messaging platform while maintaining acceptable performance levels. However, scaling complexity increases non-linearly and potential bottlenecks arise from shared resources.

4. Backlog draining

One of the essential roles of a messaging bus is to act as a buffer between different applications or platforms. When consumers are unavailable or not enough, the platform accumulates the data for later processing. In these situations, it is vital that consumers can quickly drain the backlog of accumulated data and catch up with the newly produced data. During this catch-up process, it is crucial that the performance of existing producers is not impacted in terms of throughput and latency, either on the same topic or on other topics within the cluster. This test aimed to evaluate the ability of each messaging bus to effectively support consumers in catching up with backlog data while minimizing the impact on the producer performance.

How We Set Up the Tests

We conducted all tests using the OpenMessaging Benchmark tool on AWS EC2 instances. For consistency, we utilized similar instances to test each messaging platform. Our workloads used 1KB messages with randomized payloads and a single partition per topic. We had 16 producers, and 16 consumers per subscription, with one subscription in total. To ensure durability, we configured topics to have two guaranteed copies of each message, resulting in a replica count of three. We documented any deviations from the protocol in the individual tests.

We conducted these tests at each platform's "maximum producer rate" for the outlined hardware and workload configuration. Although the OMB tool includes an adaptive producer throughput mode, this was not found to be reliable and would often undershoot or behave erratically. Instead, we adopted a manual protocol to determine appropriate producer throughput rates. For each workload, we ran multiple test instances at different rates, narrowing down the maximum attainable producer rate that resulted in no producer errors and no accumulating producer or consumer backlog. In this scenario, we could be confident that the platform would be in a steady state of near maximum end-to-end throughput. Given the discrete nature of this protocol, it is possible that real-world maximum producer rates could be slightly higher and have greater variability than those determined for the tests.

Infrastructure topology

Client instances:	4 × m5n.8xlarge
Broker instances:	3 × i3en.6xlarge

Platform versions

Apache Pulsar:	2.11.0
RabbitMQ:	3.10.7
NATS JetStream:	2.9.6

Platform-specific caveats

Pulsar – Our Pulsar setup had the broker and bookies co-located on the same VM, 3 × i3en.6xlarge topology. The ZooKeeper instance was set up separately with 3 × i3en.2xlarge topology.

RabbitMQ – We conducted tests using Quorum Queues, the recommended method for implementing durable and replicated messaging. While the results indicated that this operating mode in RabbitMQ has slightly lower performance than the "classic" mode, it offers better resilience against single-node failures.

NATS JetStream – During our tests, we attempted to follow the recommended practices for `deliverGroups` and `deliverSubjects` in NATS, but encountered difficulties. Our NATS subscriptions failed to act in a shared mode and instead exhibited a fan-out behavior, resulting in a significant read amplification of 16 times. This likely significantly impacted the overall publisher performance in the subscription count test. Despite our best efforts, we were unable to resolve this issue.

Benchmark Parameters & Results

All reported message rates are platform aggregates, not for individual topics, producers, subscriptions, or consumers.

1. Node failure

Test Parameters

In a departure from the standard test parameters, in this test we employed five broker nodes instead of three and five client nodes instead of three — two producing and three consuming. We made this change to satisfy the requirement for three replicas of a topic, even when one cluster node is absent.

In each case, messages were produced onto 100 topics by 16 producers per topic. Messages were consumed using a single subscription per topic, shared between 16 consumers.

We adopted the following test protocol: five minutes of warm-up traffic, clean termination of a single broker node, five minutes of reduced capacity operation, resumption of the terminated broker, and five minutes of normal operation. The broker was intentionally terminated and resumed using the `systemctl stop` command on the node to simulate a failure.

Test Results

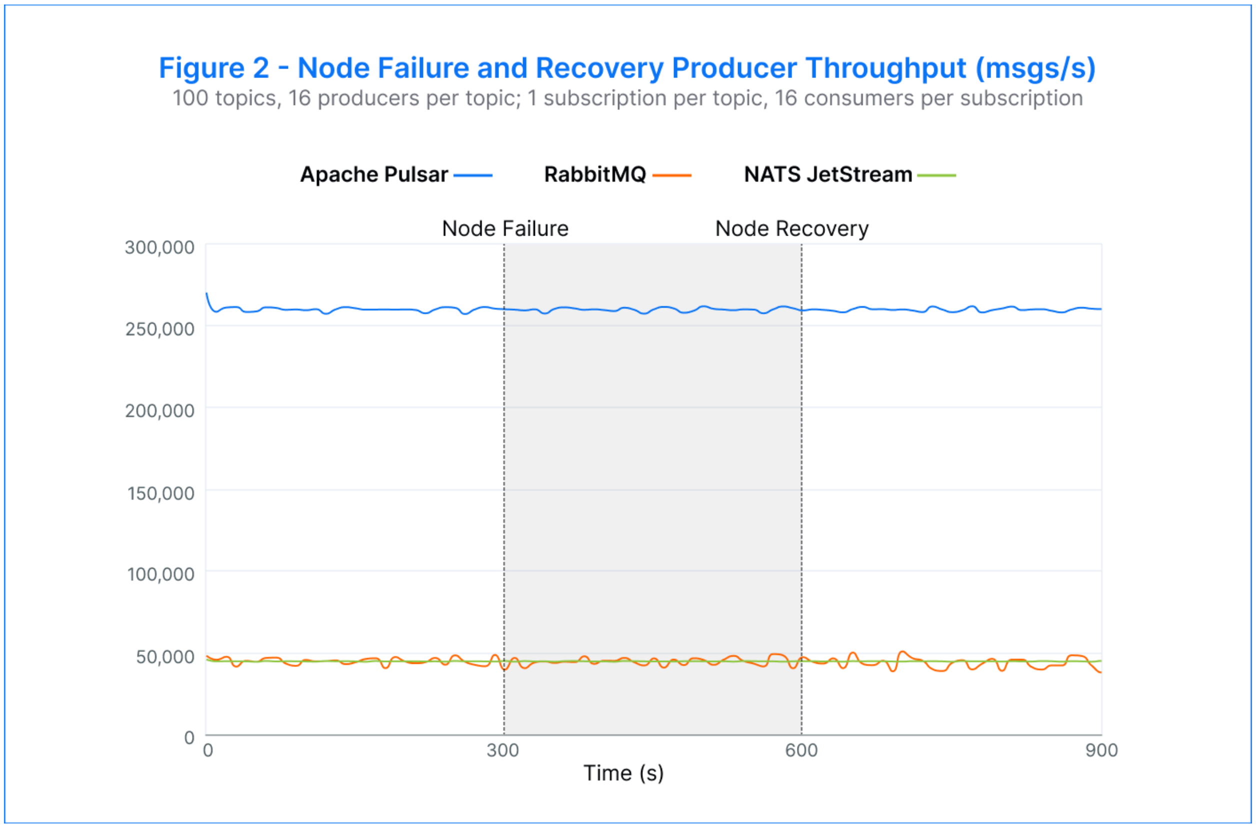

Figure 2 - Node Failure and Recovery - Producer Throughput (msgs/s)

Node State	**Average Producer Throughput (msgs/s)**		
	Apache Pulsar	**RabbitMQ**	**NATS JetStream**
Before Node Failure	260K	45K	45K
Node Failure	260K	45K	45K
After Node Recovery	260K	44K	45K

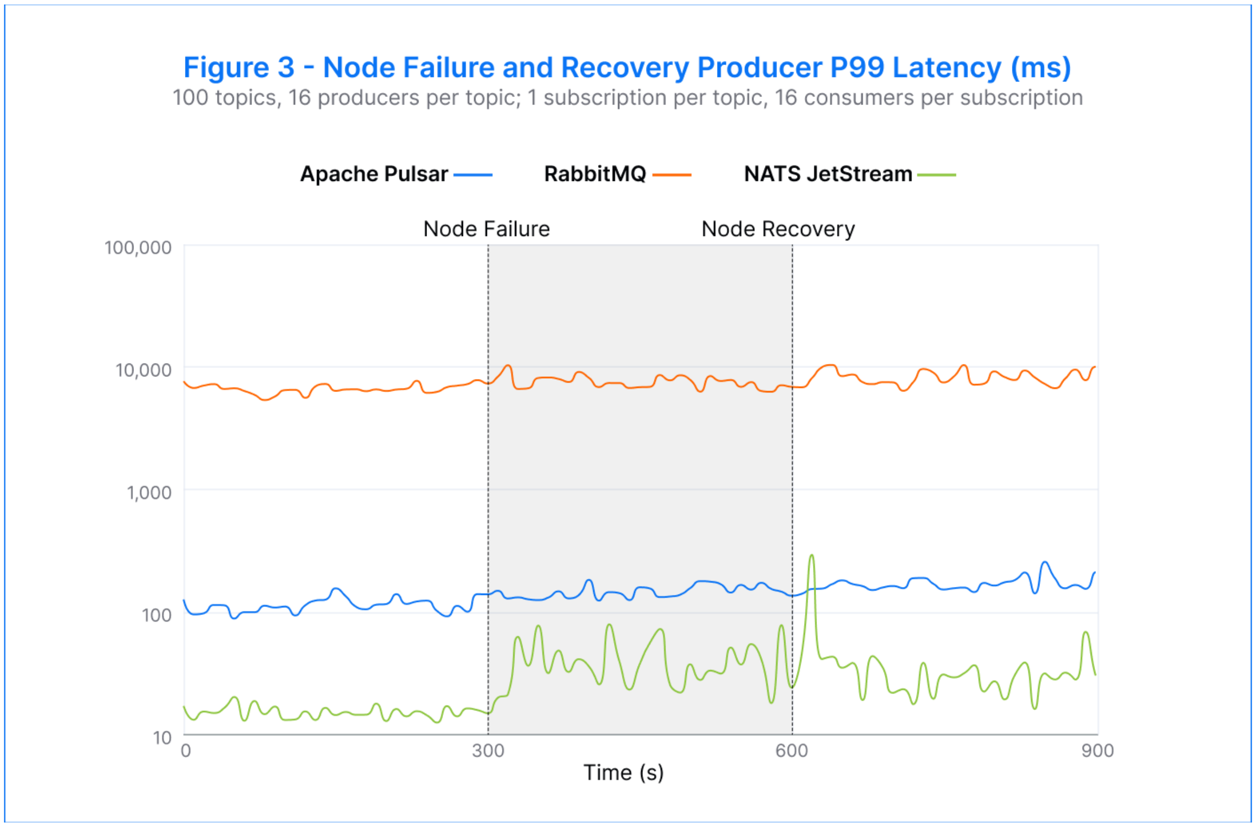

Figure 3 - Node Failure and Recovery - Producer P99 Latency (ms)

Node State	Producer P99 Latency (ms)		
	Apache Pulsar	**RabbitMQ**	**NATS JetStream**
Before Node Failure	113	6.6K	15
Node Failure	147	7.6K	40
After Node Recovery	168	8.2K	40

Pulsar – Given that Pulsar separates computing and storage, we ran two experiments to test the behavior in the event of a failed broker and a failed bookie. We consistently observed the expected publisher failover in both cases, with an average publish rate of 260K msgs/s. There was no noticeable decline in publish rate and an increase in latency from 113 milliseconds to 147 milliseconds when running on fewer nodes. Our results for both broker and bookie termination scenarios were very similar.

RabbitMQ – In the test with RabbitMQ, we noted a successful failover of producers from the terminated node, maintaining an average publish rate of 45K msgs/s. At the time of node failure, the publish latency increased from 6.6 seconds to 7.6 seconds. However, upon restart, RabbitMQ did not rebalance traffic back onto the restarted node, resulting in a degraded publish latency of 8.2 seconds. We suspect this behavior is attributed to the absence of a load balancer in the default configuration used. Nevertheless, it should be possible to implement an external load-balancing mechanism.

NATS JetStream – During the test with NATS, we observed successful failover of producers from the terminated node, with an average publish rate of 45K msgs/s. When we attempted to reach higher publish rates, however, the failover did not always occur, resulting in a corresponding increase in publish errors. The producers switched over to the alternate node within approximately 20 seconds of the broker termination. The publisher rates remained stable with minimal disruptions throughout the test. Despite this, there was an increase in p99 publish latency (as seen in Figure 3), rising from 15 milliseconds to 40 milliseconds. This latency increase persisted for the test's duration, even after the terminated broker was resumed.

Analysis

All platforms successfully transferred the work of a failed broker to other nodes and maintained the target publisher rate. It's important to note that NATS JetStream did not achieve this consistently. Both RabbitMQ and NATS JetStream showed an increase in p99 publish latency, which was expected, but they did not recover after the reintroduction of the terminated broker. This suggests that the platforms did not effectively redistribute the work to the resumed broker.

In contrast, Pulsar was the only platform that consistently and successfully transferred the work to other nodes and maintained an unaffected publish rate with a slight increase in p99 latency. Moreover, Pulsar was able to achieve an average publish rate of 260K msgs/s when running on fewer nodes, demonstrating its ability to scale efficiently even in the face of node failures.

2. Topic counts

Test Parameters

In this test, we ran multiple tests on each platform, varying the number of independent topics in each instance and measuring the publish throughput and latency.

Test Results

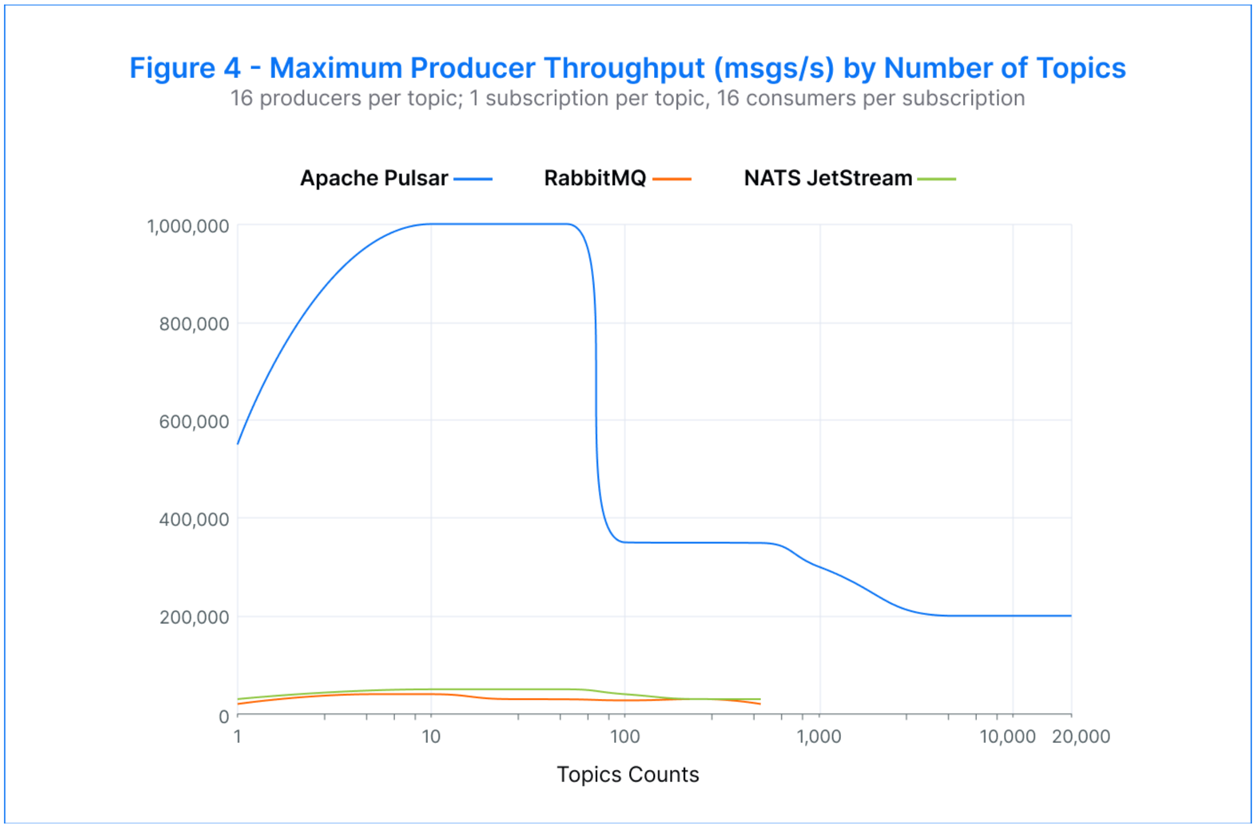

Figure 4 - Maximum Producer Throughput (msgs/s) by Number of Topics

Number of Topics	Maximum Producer Throughput (msgs/s)		
	Apache Pulsar	RabbitMQ	NATS JetStream
1	550K	20K	30K
10	1.0M	40K	50K
50	1.0M	30K	50K
100	350K	27K	40K
500	350K	20K	30K
20,000	200K	N/A	N/A

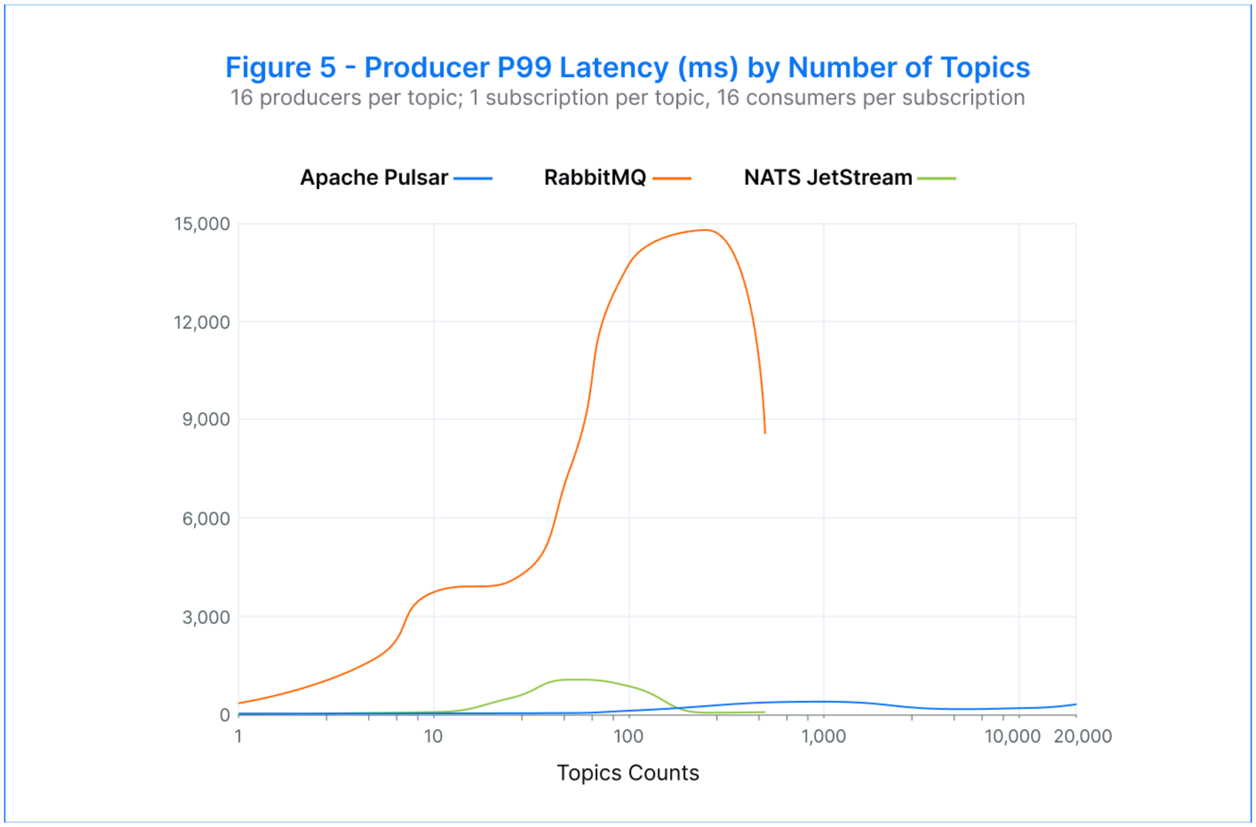

Figure 5 - Producer P99 Latency (ms) by Number of Topics

Number of Topics	Producer P99 Latency (ms)		
	Apache Pulsar	RabbitMQ	NATS JetStream
1	7	344	22
10	14	3,734	75
50	24	7,502	1,064
100	93	13,740	866
500	-	8,565	72
20,000	293	N/A	N/A

Pulsar – The platform achieved an aggregate publisher throughput of 1M msgs/s with a topic count between 10 and 50. Across thousands of topics, Pulsar maintained low publisher p99 latency, ranging from single-digit milliseconds to low hundreds of milliseconds (~7 ms to 300 ms).

We can see from the chart that there was a negative inflection point in the throughput when the number of topics exceeded 100. This variation can be attributed to the effectiveness of batching at different topic counts:

- With fewer topics, the throughput per topic is relatively high and it conducts for a very high batching ratio (messages/batch). This means that it's very efficient to move a large number of messages through the platform, in a small amount of batches. In these conditions, the bottleneck is typically on the I/O system.
- With more topics, we are spreading the throughput over a larger number of them. The per-topic throughput is therefore lower and the batching ratio decreases, until we end up with just one message per batch. At this point, the bottleneck has shifted to the CPU cost instead of the I/O system.

RabbitMQ – The publisher throughput fluctuated between 20K and 40K msgs/s across the range of topics. Meanwhile, the p99 publish latency rose significantly. These latencies often exceeded multiple seconds, ranging from 344 milliseconds to nearly 14 seconds. Testing was stopped after 500 topics as it became challenging to construct the topics in a reasonable amount of time.

NATS JetStream – The best performance was observed when using 10 to 50 topics, with a rate of 50K msgs/s. As the number of topics increased beyond 50, the throughput gradually decreased. The p99 publisher latencies also started to increase, starting from 75 milliseconds at 10 topics to over one second at 100 topics. The testing was stopped at 500 topics due to the difficulty in constructing additional topics, but the system could still handle 30K msgs/s at this configuration.

Analysis

The results suggest that all of the platforms tested could handle larger topic counts in real-world scenarios where topics accumulate gradually over time, rather than the time-consuming process of generating test topics. Despite this, RabbitMQ and NATS JetStream demonstrated a performance decline when concurrently publishing a very large number of topics.

On the other hand, Pulsar outperformed RabbitMQ and NATS JetStream in the number of topics, publish rate, and latency. The results show that Pulsar could handle 10 times more topics. Pulsar achieved up to 1M msgs/s, surpassing RabbitMQ by 33 times and NATS JetStream by 20 times. Pulsar also demonstrated exceptional latency performance, with p99 latency 300 times better than RabbitMQ and 40 times better than NATS JetStream at 50 topics. Pulsar was able to maintain producer throughput of 200K msgs/s at 20K topics.

3. Subscription counts

Test Parameters

In this test, we expected a significant boost in reads with a larger number of subscribers. To achieve this, we limited the number of concurrent topics to 50, assigned a single consumer to each subscription, and set a minimum aggregate publish rate of 1K msgs/s for the platform.

Test Results

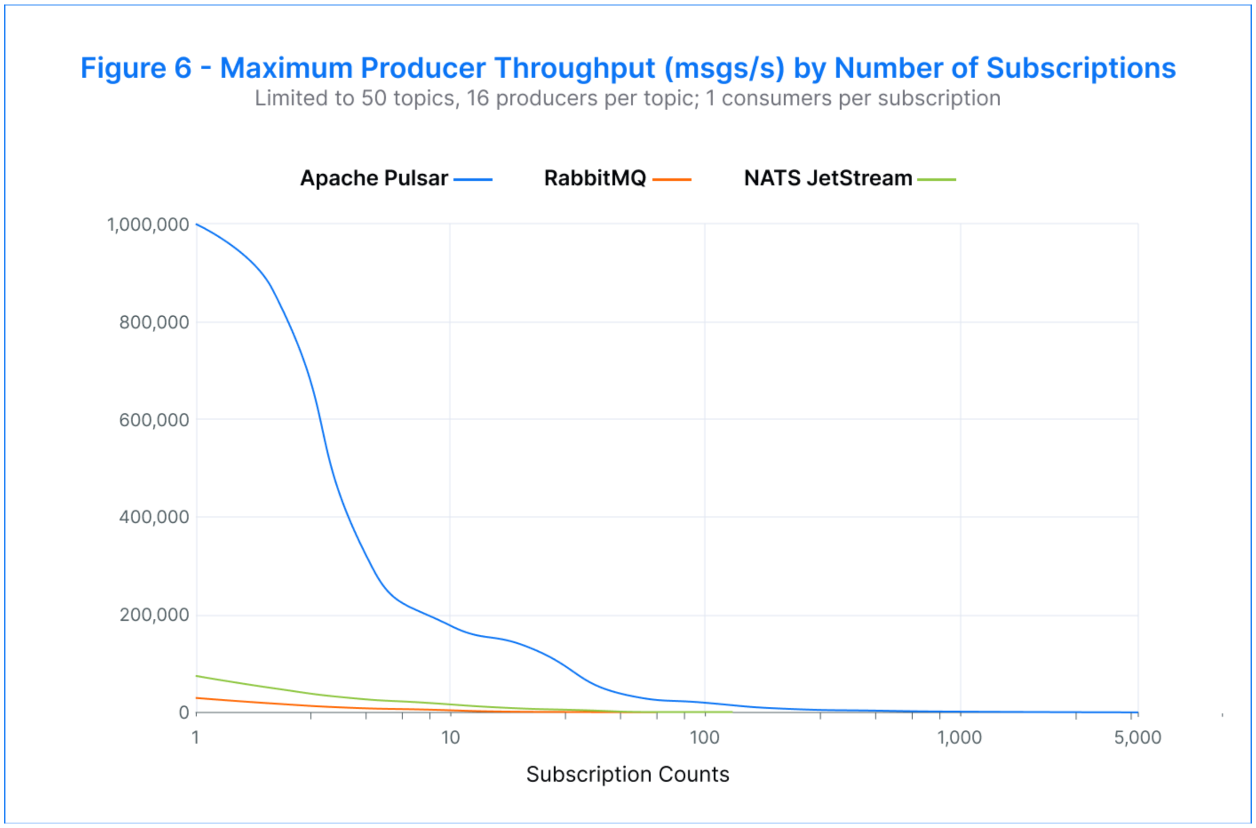

Figure 6 - Maximum Producer Throughput (msgs/s) by Number of Subscriptions

Number of Subscriptions	Maximum Producer Throughput (msgs/s)		
	Apache Pulsar	RabbitMQ	NATS JetStream
1	1M	30K	75K
10	178K	3.5K	-
50	35K	705	2K*
100	20K	N/A	1K*
5,000	359	N/A	N/A

**Estimate based on subscription = 32, 64, and 128*

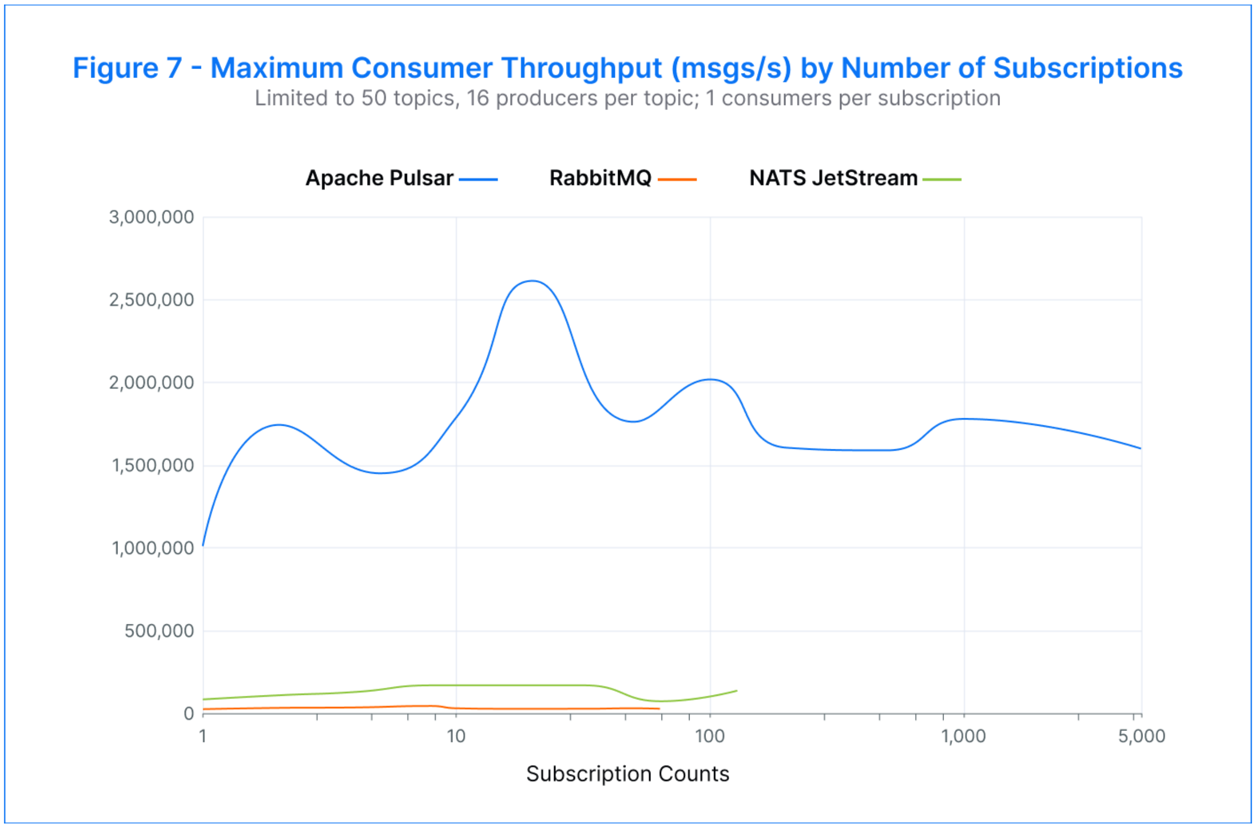

Figure 7 - Maximum Consumer Throughput (msgs/s) by Number of Subscriptions

Number of Subscriptions	Maximum Consumer Throughput (msgs/s)		
	Apache Pulsar	RabbitMQ	NATS JetStream
1	1.0M	30K	75K
10	1.8M	35K	160K*
20	2.6M	35K	160K*
50	1.8M	35K	-
100	2.0M	N/A	-
5,000	1.6M	N/A	N/A

**Estimate based on subscription = 16, 32, and 64*

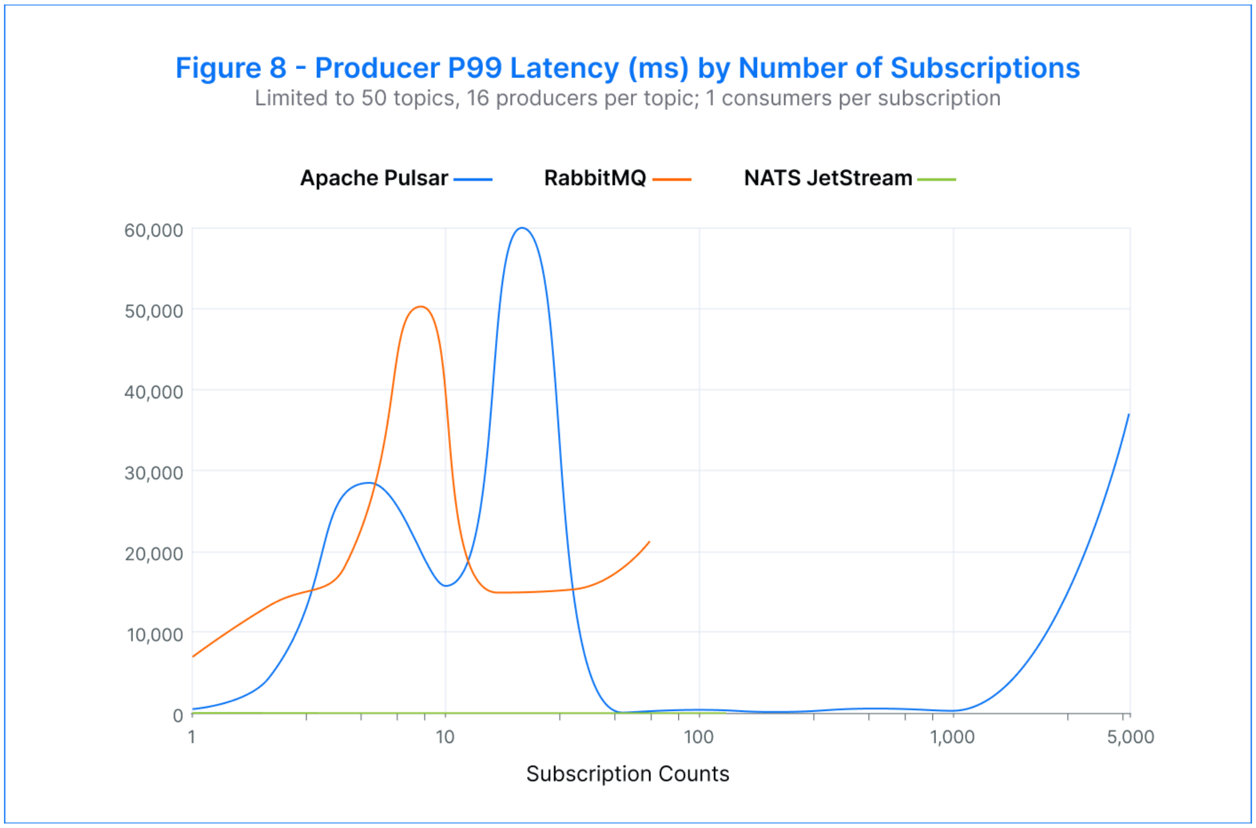

Figure 8 - Producer P99 Latency (ms) by Number of Subscriptions

Number of Subscriptions	Producer P99 Latency (ms)		
	Apache Pulsar	RabbitMQ	NATS JetStream
1 - 50	Up to 60K	Up to 50K	Up to 34
50 - 100	Up to 413	Up to tens of seconds	Up to 3
100 - 1,000	Up to 572	N/A	Up to 30
1,000 - 5,000	Up to 37K	N/A	N/A

Pulsar – We were able to achieve a maximum of 5K subscriptions per topic before consumers started to fall behind. However, even with higher subscription numbers, the publish latency remained low. In fact, we measured peak consumer throughput at an impressive 2.6M msgs/s. During our test, we identified an issue with many concurrent I/O threads competing for the same resource. However, we were able to address this in Pulsar version 2.11.1. For more information on this issue, please refer to the GitHub PR #19341.

RabbitMQ – The maximum number of successful subscriptions per topic achieved was 64. Beyond that, the publish rate dropped to around 500 msgs/s, and the p99 publish latency increased significantly to tens of seconds. Additionally, the clients became unresponsive beyond 64 subscriptions. However, the aggregate consumer throughput remained around 35K msgs/s and reached a peak of 48K msgs/s when there were eight subscriptions.

NATS JetStream – We achieved a maximum of 128 subscriptions per topic. As the number of subscriptions increased, there was an increase in publisher errors and lagging consumers. Despite this, the publish latency remained consistently low, ranging from 3 milliseconds to 34 milliseconds across all subscriptions. The highest consumer throughput was recorded at 160K msgs/s during eight to 32 subscriptions.

Analysis

As expected in this test case, end-to-end throughput became limited by the consumer. Pulsar was able to support hundreds of subscriptions per topic while maintaining very low publish latency. RabbitMQ and NATS JetStream achieved fewer subscriptions, and RabbitMQ experienced a significant increase in publish latency as the number of subscriptions increased. Pulsar stood out as the most efficient platform, demonstrating a publish rate and an aggregate consumer throughput that were both an order of magnitude higher than the other platforms.

4. Backlog draining

Test Parameters

In this test, the conditions were set to generate a backlog of messages before consumer activity began. Once the desired backlog size was reached, consumers were started, and messages continued to be produced at the specified rate. The backlog size was set to 300GB, larger than the available RAM of the brokers, simulating a scenario in which reads would need to come from slower disks rather than memory-resident caches. This was done to evaluate the platform's ability to handle catch-up reads, a common challenge in real-world scenarios.

During the tests, messages were produced on 100 topics, with 16 producers per topic. Messages were consumed using a single subscription per topic, shared between 16 consumers.

Test Results

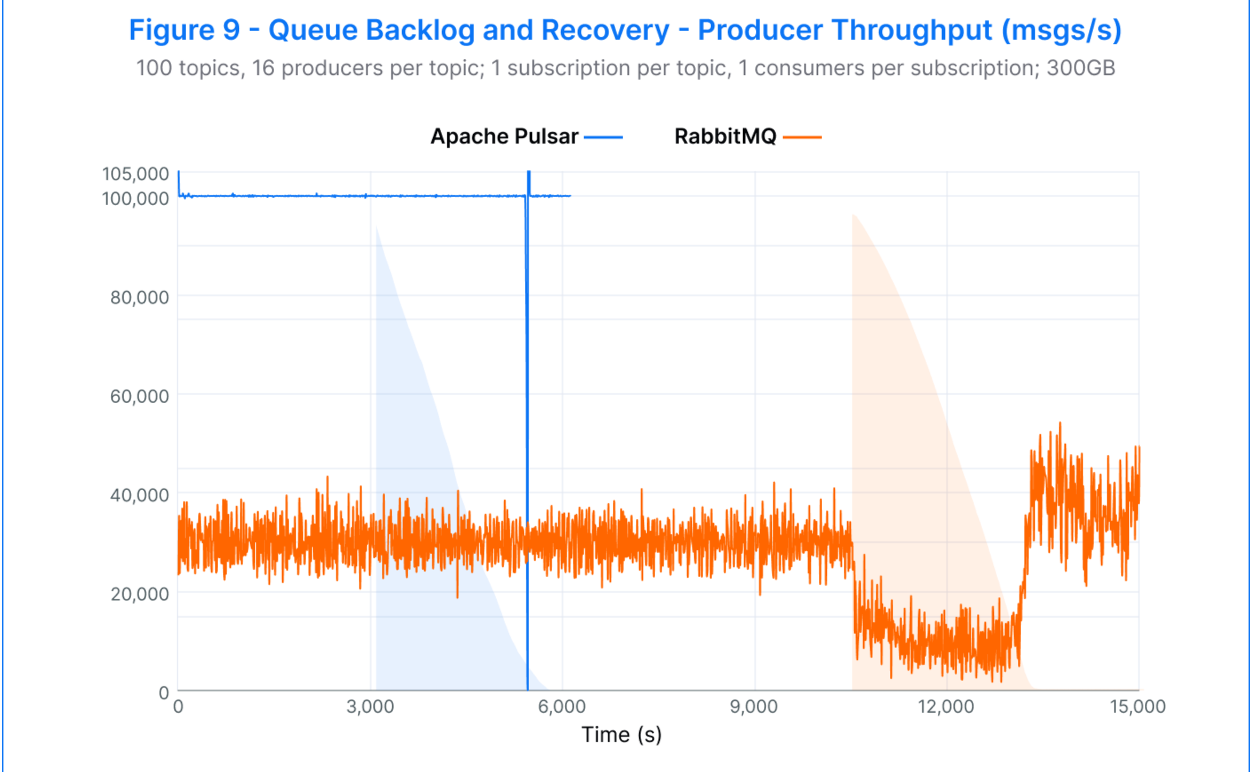

Figure 9 - Queue Backlog and Recovery - Producer Throughput (msgs/s)

300GB Queue Backlog	Average Producer Throughput (msgs/s)	
	Apache Pulsar	RabbitMQ
Pre-Drain	100K	30K
Drain	100K	12.5K
Post-Drain	100K	37K

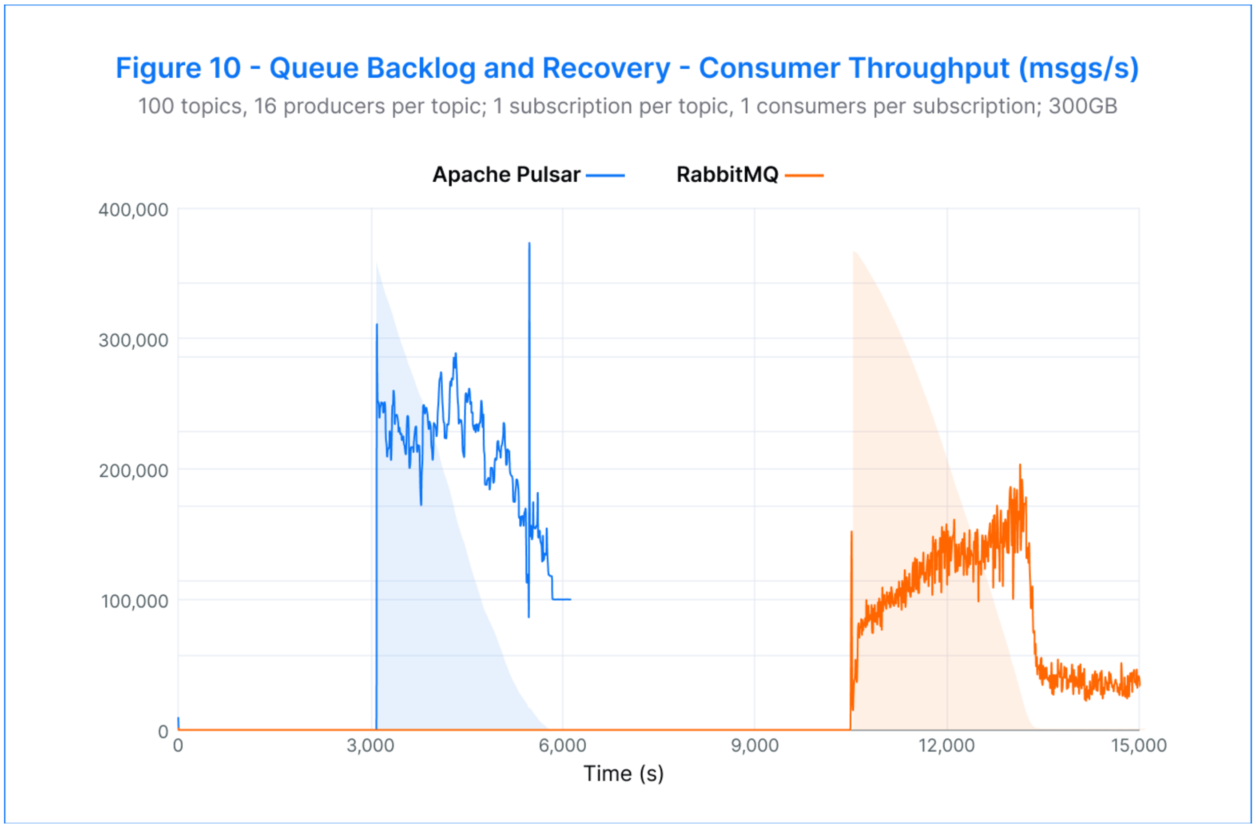

Figure 10 - Queue Backlog and Recovery - Consumer Throughput (msgs/s)

300GB Queue Backlog	Average Consumer Throughput (msgs/s)	
	Apache Pulsar	RabbitMQ
Pre-Drain	100K	30K
Drain	213K	121K
Post-Drain	100K	37K

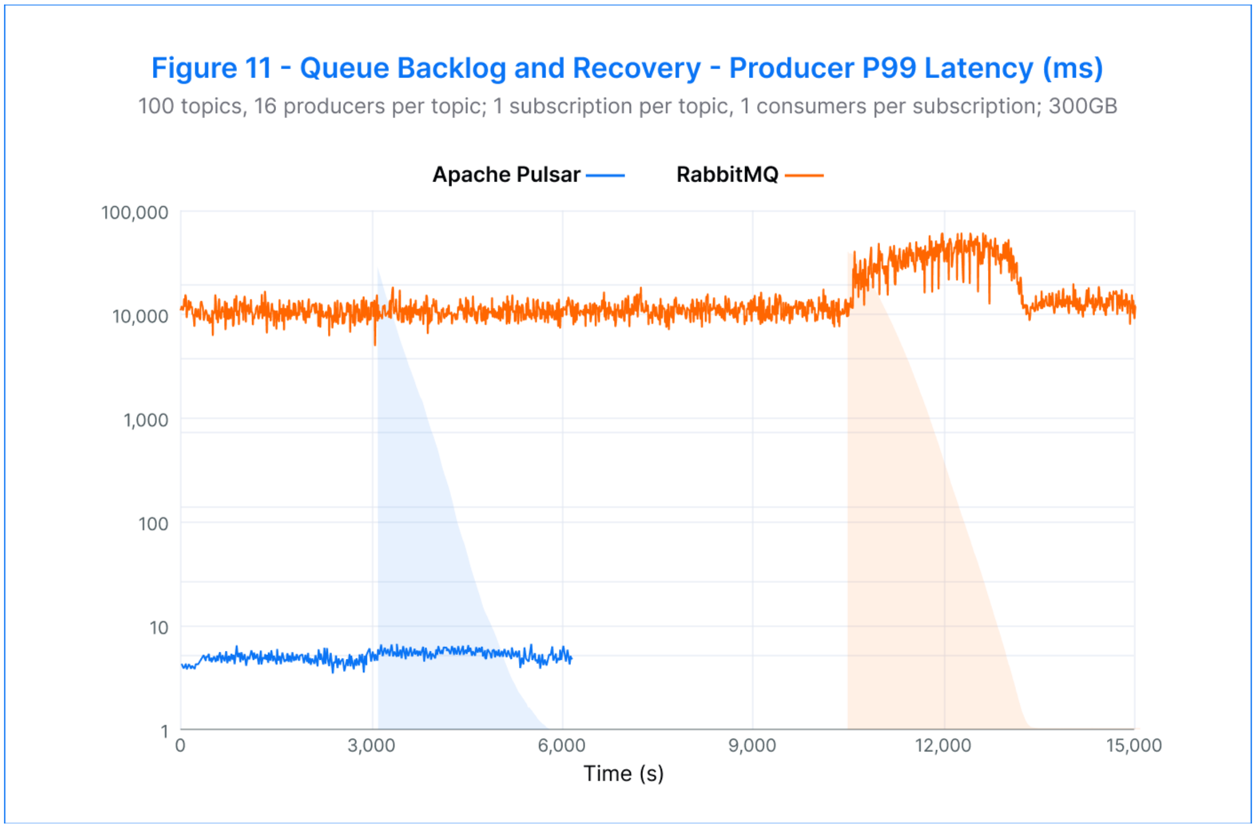

Figure 11 - Queue Backlog and Recovery - Producer P99 Latency (ms)

300GB Queue Backlog	Average Producer P99 Latency (ms)	
	Apache Pulsar	**RabbitMQ**
Pre-Drain	4.7	11K
Drain	5.3	34K
Post-Drain	4.9	13K

Pulsar – In this test, Pulsar delivered impressive results in terms of producer and catch-up read rates. The producer rate remained stable at 100K msgs/s before, during, and after the drain, and catch-up reads averaged 200K msgs/s. The drain itself was completed in approximately 45 minutes.

During the backlog drain phase, a slight increase in p99 publish latency from 4.7 milliseconds to 5.3 milliseconds was observed. However, this was expected due to the increased contention between producers and consumers.

One of the most noteworthy findings of the test was that Pulsar's consumer throughput returned to its pre-drain level after the drain was complete. This showcased Pulsar's ability to handle high volumes of data without compromising performance.

RabbitMQ –RabbitMQ was able to achieve its target producer rate of 30K msgs/s, but the platform faced a challenge when reads dominated during backlog production, leading to a steal of IOPS and hindering message production. This resulted in a reduction of the producer rate to 12.5K msgs/s, with a latency increase of three times from 11 to 34 seconds. However, the catch-up reads were swift, starting at 80K msgs/s and steadily rising to 200K msgs/s. After 50 minutes, most of the backlog had been drained, and the producer throughput was regained, with the latency returning to approximately 13 seconds. Despite a consistent yet small consumer backlog, the platform remained stable.

NATS JetStream – Unfortunately, NATS could not produce any results in this test. The clients encountered OOM errors while building the backlog, which we suspect might be due to a potential issue in the jnats library.

Analysis

Pulsar demonstrated impressive producer and catch-up read rates during the test, with stable performance before, during, and after the drain. Pulsar's consumer throughput returned to its pre-drain level, showcasing its ability to handle high volumes of data without compromising performance. Pulsar also outperformed RabbitMQ by being 3.3 times faster in producing and consuming, and the drain would have been completed even faster if Pulsar had been set to a 30K msgs/s producer rate.

RabbitMQ demonstrated some impressive consumer rates when reading the backlog. However, this came at the cost of message production, as the consumers had clear priority. In a real-world scenario, applications would be unable to produce during the catch-up read and would have to either drop messages or take other mitigating actions.

It would have been interesting to see how NATS JetStream performed in this area, but further work will be needed to investigate and resolve the suspected client issue.

Conclusion

The benchmark tests showed that Pulsar can handle significantly larger workloads than RabbitMQ and NATS JetStream and remain highly performant in various scenarios. Pulsar proved its reliability in the presence of node failure and its high scalability for both topics and subscriptions. Conversely, RabbitMQ and NATS JetStream both showed a decline in performance when concurrently publishing a large number of topics.

The results suggest that while all three platforms are suitable for real-world scenarios, it is crucial to carefully evaluate and choose the technology that best aligns with the specific needs and priorities of the application.

Key findings summarizing Pulsar's performance:

1. Pulsar maintained high publish rates despite broker or bookie failure. No degradation in rates occurred when running on fewer nodes, with 5 times greater maximum publish rates than RabbitMQ and NATS JetStream.
2. Pulsar achieved high performance with 1M msgs/s, surpassing RabbitMQ by 33 times and NATS JestStream by 20 times. With a topic count of 50, p99 latency was 300 times better than RabbitMAQ and 40 times better than NATS JetStream. Pulsar was able to maintain a producer throughput of 200K msgs/s at 20K topics. In contrast, RabbitMQ and NATS JetStream failed to construct topics beyond 500 counts.
3. Pulsar supported 1,024 subscriptions per topic without impacting consumer performance, while maintaining low publish latency and achieving a peak consumer throughput of 2.6M msgs/s. This was 54 times faster than RabbitMQ and 43 times faster than NATS JetStream.
4. Pulsar achieved stable publish rates and an average catch-up read throughput of 200K msgs/s during the backlog drain test case. In comparison, RabbitMQ's publish rate dropped by over 50 percent during draining and resulted in an increase in publish latency by three times.

RabbitMQ may be a suitable option for applications with a small number of topics and a consistent publisher throughput, as the platform struggles to deal with node failures and large backlogs. NATS may be a good choice for applications with lower message rates and a limited number of topics (less than 50). Overall, the results show Pulsar outperforms RabbitMQ and NATS JetStream in terms of throughput, latency, and scalability, making Pulsar a strong candidate for large-scale messaging applications.

Want to Learn More?

For more on Pulsar, check out the resources below.

1. Learn more about how leading organizations are using Pulsar by checking out the latest Pulsar success stories.
2. Use StreamNative Cloud to spin up a Pulsar cluster in minutes. Get started today.
3. Engage with the Pulsar community by joining the Pulsar Slack channel.
4. Expand your Pulsar knowledge today with free, on-demand courses and live training from StreamNative Academy.

About StreamNative

StreamNative is a messaging and streaming platform powered by Apache Pulsar and built for cloud-native, event-driven applications. Founded in 2019 by the original creators of Pulsar, StreamNative has more experience designing, deploying, and running large-scale Apache Pulsar instances than any other team in the world. With StreamNative Cloud, you get a fully managed Pulsar-as-a-Service offering available in our cloud or yours. Learn more at streamnative.io.

References

[1] Comparing Pulsar and Kafka: Unified Queuing and Streaming:
https://www.splunk.com/en_us/blog/it/comparing-pulsar-and-kafka-unified-queuing-and-streaming.html
[2] The Linux Foundation Open Messaging Benchmark suite:
http://openmessaging.cloud/docs/benchmarks
[3] The Open Messaging Benchmark Github repo:
https://github.com/openmessaging/benchmark
[4] GitHub Pull Request #19341:
https://github.com/apache/pulsar/pull/19341